DOCS Design Document

Part - A

Prepared by:
Sourodeep Datta,
Tanishq Prasad,
&
Sanskar Mittal

Contents

Chapter 1 Database Design

1.1 Optimized Workload

We have implemented the database optimizing for read-intensive workloads.

1.2 Data Structures Used

In order to implement the database, we have implemented a file-persistent HashMap.

1.2.1 Persistent HashMap

Given the nature of the data being stored—key-value pairs with variable-length string keys—we determined that a hashmap would be the most suitable choice for our storage engine. While B-trees are commonly used for such tasks, their existing implementations are typically optimized for integer keys or fixed-length keys.

Creating an efficient, file-persistent B-tree implementation for variable-length string keys would involve significant complexity. This would require either imposing a maximum size on the keys or employing prefix-trees for handling strings efficiently, both of which add overhead. Additionally, the overall complexity of implementing and maintaining such a solution outweighed its potential benefits for our use case.

Instead, we opted for a hashmap to map variable-length string keys to bins, leveraging its simplicity and efficiency for read-heavy workloads. The hashmap approach allows for dynamic key handling without predefined constraints, making it well-suited for our application's requirements.

1.2.2 Hashing

The hashmap in our storage system relies on the standard std::hash function from the C++ Standard Library to compute hash values for the string keys. This function is defined in the <functional> header.

1.2.3 Bins

In our storage system, a bin serves as the persistent storage unit for key-value pairs. Each bin is implemented as a binary file on disk, containing a subset of the dataset as determined by the hashmap's hash function.

Bin Structure

Each bin stores key-value pairs in a sequential binary format. The structure of each entry is as follows:

Field	Size	Description
Key Length	4 bytes (`int`)	Length of the key in bytes.
Value Length	4 bytes (`int`)	Length of the value in bytes.
Deleted Flag	1 byte (`bool`)	Indicates whether the key is marked as deleted.
Key	Variable	The actual key data, stored as a variable-length string.
Value	Variable	The actual value data, stored as a variable-length string.

Table 1.1: Structure of a Bin Entry

Bin Operations

- **Insertion (SET):**
  When a key-value pair is inserted into the system, it is written to the corresponding bin based on the hash function. It appends key-value pair the bin file in the specified format to the end of the file. If the key already exists in the file, it marks the corresponding entry as deleted before appending.

- **Retrieval (GET):**
  To retrieve a value for a given key, the file is sequentially scanned find the matching key, and it's corresponding value is returned.

- **Deletion (DEL):**
  To delete a key, the entry in the file is located, and its deleted flag is set to true. This avoids rewriting the entire file, making deletions efficient.

  Deferred Cleanup: As deletions do not immediately reclaim space in the file, the space has to be reclaimed during a periodic defragmentation process, which rewrites the bin file, excluding deleted entries.

1.2.4 Storage Engine

The StorageEngine class acts as a high-level interface for managing persistent storage operations in the system. It abstracts away the underlying implementation details of the HashMap, providing a clean and user-friendly API for interacting with the storage backend. To achieve this, the Pointer-to-Implementation (PImpl) idiom is used, which separates the interface from the implementation, improving encapsulation and flexibility.

1.3 Concurrency Control

Given the read-heavy nature of the workload, the concurrency model is designed to maximize parallelism for read (GET) operations while maintaining strict control during write (SET, DEL) operations. This ensures a balance between performance and correctness.

The storage system employs reader-writer synchronization techniques to manage access to shared resources (bins). The strategy focuses on optimizing for multiple readers while ensuring exclusive

access for writers. This system relies on std::shared_mutex, a standard C++ synchronization primitive, to implement these locks effectively.

1.3.1 Granularity

Each bin is treated as an independent unit, thus locks are implemented at the bin level instead of globally.

1.3.2 Locking Protocol

- Read operations acquire a shared lock, enabling multiple threads to access the bin.
- Write operations acquire an exclusive lock, temporarily blocking all other threads from accessing the bin.

1.4 Caching

In our system, caching allows rapid retrieval of key-value pairs from memory, avoiding the need for disk I/O operations on every access. The system uses an in-memory cache to hold recently accessed data, while the persistent storage layer manages data durability and consistency.

1.4.1 Cache Structure

The cache is bin specific, with the number of entries determined during compile time.

The cache is organized as an LRU (Least Recently Used) cache, which evicts the least recently accessed entries when space is full. The LRU algorithm ensures that the most frequently accessed keys remain in memory, improving the chances of a cache hit for future lookups.

1.4.2 Cache Lookup

When a key is requested, the system first checks if it exists in the cache.

If found, the value is returned directly from memory, avoiding the overhead of disk I/O.

If the key is not found, a cache miss occurs, and the system retrieves the key from the persistent storage, updating the cache with the new data.

1.4.3 Cache Updation

The cache employs a write-through policy to ensure that any updates to the data in the cache are immediately reflected in the persistent storage. This ensures consistency between the in-memory cache and the underlying storage layer.

1.4.4 Cache Consistency

Spinlocks are used to ensure thread safety when multiple threads attempt to access or modify the cache concurrently.

1.5 Garbage Collector

Garbage collection (GC) in the storage system is responsible for ensuring that deleted or obsolete entries do not occupy unnecessary space in the bin files. Over time, as keys are added and deleted, gaps may form in the bin files, leading to inefficient disk usage. The garbage collector periodically reclaims this wasted space, ensuring that the system remains efficient in both storage and performance.

1.5.1 Garbage Collection Process

The garbage collection process is designed to run in the background, periodically checking and compressing bin files to remove entries marked as deleted. This process includes:

- **File Scanning:**
  The garbage collector starts by scanning the bin files for entries marked as deleted. These entries are skipped during the read/write operations but still consume space.
- **Entry Validation:**
  Each entry in the file is checked for validity. If an entry is marked as deleted, it is skipped. Otherwise, the valid data is retained for the next operation.
- **File Recompression:**
  After identifying and skipping deleted entries, the garbage collector rewrites the valid entries back into the bin file. This process reduces fragmentation and ensures that only valid data remains in the file.

The garbage collector runs with a write lock on each bin. This ensures that no other thread can modify the bin while it is being compressed, maintaining data integrity during the cleanup process.

1.5.2 Garbage Collection Trigger

The garbage collector runs periodically at fixed intervals. In this system, the interval is determined by a constant GARBAGE_COLLECTOR_INTERVAL, set to a specific duration (e.g., every 30 seconds).